A Portable Logic Simulation System

AUTHOR(S):

Shimizu, Kentaro

数理解析研究所講究録
511 巻 1984 年 144-155

144

A Portable Logic Simulation System

Kentaro Shimizu

Department of Information Science, University of Tokyo,
Bunkyoku, Tokyo, 113 Japan

SUMMARY

This paper describes a portable logic simulation system PLS, which was used for development of FLATS -- a formula manipulation machine consisting of more than 33,000 ECL and partly TTL MSI chips. PLS supports two simulation languages, HDL(Hardware Description Language) and SCL(Simulation Control Language). Register-transfer and/or gate level design specifications are written in HDL. SCL describes control information about the simulation. Both descriptions are translated into Fortran and are linked to execute the simulation. The PLS system is implemented mainly in Fortran. Fortran was used for portability and efficiency. PLS checks for several types of illegal specifications globally at compile time and executes one-pass simulation; thereby the execution time is considerably shortened. The system covers a wide area of application and its conciseness facilitates expressing, organizing, and in general, dealing with large digital systems. This paper also describes a preprocessor RATFOR-LS, which is an extension of RATFOR in bit-manipulating operations to facilitate describing and simulating computer hardware.

1. INTRODUCTION

Many logic simulation systems have been proposed and implemented[1,2,7]. Very elaborate systems seem to be used in computer industry. However, such systems are of proprietary nature and are not in the public domain. Therefore, we implemented our own system, PLS to design a formula manipulation machine FLATS[3], which consists of more than 33,000 ECL and partly TTL MSI chips.

Wide area of applications, conciseness, and high efficiency in running simulations are basic requirements to be met to develop large digital systems. The PLS system takes in register-transfer and/or gate level design specifications, executes simulations, checks out illegal connections, provides special forms of documents, generates test data, and performs a variety of other tasks related to the hardware design. In PLS two simulation languages, HDL and SCL, are provided. The design specifications of the computer hardware are written in HDL. SCL describes control information about the simulation. Since the control information is separated from the hardware description, simulation conditions can be changed without modifying and recompiling the hardware description. The PLS design database maintains a description of hardware elements and linkage information used to resolve intermodule references. It provides modular and systematic organization of large digital systems. As for efficiency, by restricting the object logic to a synchronous system that has no loop structures between flipflops, every hardware operation is executed only once in each clock cycle, and the HDL compiler checks for timing errors and several types of illegal connections (discussed later) at compile time; thereby, the execution time of simulation is considerably shortened. By using the linkage information in the design database, the compiler can detect these errors globally in intermodule communications. This facility also helps the user to find errors in an early stage of the design and improves reliability of the design verification.

The FLATS project was developed by three groups, University of Tokyo(UT), the Institute of Physical and Chemical Research(IPCR), and Mitsui Engineering and Shipbuilding Co. Ltd.(MES), in collaboration. The hardware designers, the users of PLS, were scattered among these three groups, and each group had in-house machines of different architectures, as shown in Table I. The first requirement imposed on the PLS system was that it should be commonly available to the designers on their own machines. In other words, portability was the internal requirement of the project.

Table I. Implementations of the PLS system

	project group	Distance from UT	machine (architecture)	operating system	Number of users
I.	UT	-	HITAC 8700 (similar to IBM/360)	OS7	6
II.	IPCR	20km	FACOM M380 (IBM/370 compatible)	OS IV/F4	3
III.	MES	600km	VAX-11 780	VAX/VMS	3

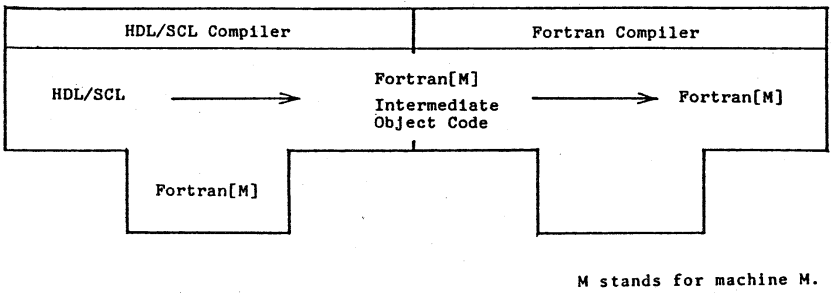

Figure 1. Translation scheme of the PLS system

Fortran was the only programming language that was commonly available on the three machines. However, we found it very difficult to describe and simulate such a large machine as FLATS by Fortran code as discussed later. We therefore designed new simulation languages, HDL and SCL, and implemented their compilers. Fortran was used as an implementation language of the PLS system and as an intermediate object language of HDL and SCL. Figure 1 shows this translation scheme.

The first design of the compilers was completed within a month. The entire system was implemented in half a year, and has been installed and used successfully on the three machines.

2. CHOICE OF LANGUAGES

2.1. Why Fortran for intermediate code?

In general, use of the intermediate code increases portability of the compiler. It facilitates transportation of the compiler to other machines. In PLS, the compilers generate intermediate code in Fortran. The translation of the code, which is usually a machine-dependent, final phase of compilation, is made by conventional Fortran compilers. Therefore, the need for writing the translator for the intermediate code is eliminated. The generated code uses only primary Fortran facilities; logical IF, GO TO, CALL, and assignment statements; arithmetic expressions and function calls. All the computations associated with the hardware operations are performed on values of integer (INTEGER*4) data type except for logical expressions in the condition of IF statements. The generated code indeed has been run essentially without change on all machines of the project groups.

The only exception is the bitwise operation with logical operators, which is used for speed up of execution. Fortran logical operators .AND., .OR., and .NOT. can be used for 32-bit bitwise operations in all the systems available. In system II, however, since the logical operators can not be applied to operands of integer data type, their operations are translated into function references. Bit manipulations have not been defined in the Fortran standards due to their dependency on the implemented machine, and now depends on Fortran dialects. In PLS, the switch of this code generation can be specified with the compiler option.

Another intention to use Fortran was efficiency. Fortran was the most efficient language (particularly for programs requiring much computations) among those that were available on each machine. Its compilers produce highly optimized object codes.

2.2. Why Fortran for system implementation language?

The fundamental reason Fortran was used for the system implementation language was that because it was the only programming language commonly available to the hardware designers on their own machines; as we have mentioned, portability was the internal requirement of our project. A second major factor is efficiency of Fortran in record level input/output, arithmetic operations, and string processing. The PLS system includes a variety of application domains such as language translation, database manipulation, numerical computation and so on. We had to consider how well the facilities in a language match those that we feel are important in each domain. The system could be written in other programming languages, such as Lisp, C and PL/I, which provide more convenient control and data structures than Fortran for programming and describing programs. However, they would be too slow to run the entire system in our environment. Fortran, because of its flexibility and lack of restrictions, was enough available in our applications without degradations and loss of portability.

During the system development, in order to reduce additional re-coding and maintenance efforts required for its transportation, we did not use any special features that were provided only by some particular compilers. However, some machine dependent coding was needed due to the difference of byte addressing scheme between IBM (I and II) and DEC (III) architectures.

Besides Fortran assembly languages were used for efficiency in some basic routines. In our original design on system I, the design database was implemented using indexed sequential i/o, and its interfacing routines were described by some 2,000 lines of the assembly code. But, in systems II and III, simple sequential i/o was used, and the routines were written in Fortran for portability reasons. The run-time routines which perform bitwise operations were written in the assembly languages on the three machines. However, they consist of about 100 lines and was easily implemented on all the machines.

2.3. Why not Fortran for the simulation language?

At the outset, some parts of FLATS machine were described and simulated by Fortran code. However, we realized and encountered many difficulties which were caused by Fortran deficiencies. It was necessary to develop a new simulation language suited for hardware description. For this reason, we designed simulation languages, HDL and SCL, and implemented their compilers. The HDL and SCL provide features that were required for describing and simulating a large digital system. The difficulties in using Fortran and the features provided by the HDL and SCL are discussed below.

First, FLATS consists of more than 33,000 MSI chips. It was almost impossible to maintain these hardware components by using Fortran variables for the following reasons:

(a) The limit of six character names of Fortran was insufficient for identifying the hardware components with sufficient names. Some symbolic name dictionary was required to manage their names.

(b) External variables, which were referred beyond the subroutine level, were declared as COMMON variables. However, in Fortran the COMMON variables are accessed by their position relative to the beginning of the COMMON block in which they lie. The resulting programs were unreliable, unless the programmers exercised great care.

In PLS, the compilers automatically generate a unique Fortran name for the PLS identifier, the first twelve characters of which are significant. The original name and the associated Fortran name are stored in the design database. It also contains linkage information used to resolve external references.

Second, in manipulating bytes and words in computer hardware, it was often necessary to access a field of bits in a byte or a word. In Fortran, however, there are no facilities provided for inserting and extracting a value to a bit field. The routines that perform these operations had to be written in the assembly language, and the resulting code therefore was particularly difficult to read and understand. Several bitwise operations were also described as function references.

As for bit processing in Fortran, another internal form of binary data (0 or 1) might be used; to unpack contiguous bits into a Fortran integer array of so many elements, each containing the binary data. This method however requires excessive storage and computation time. The special dedicated routines would have to be written for arithmetic and relational operations on the unpacked data as well as for bitwise operations. The PLS compilers provide facilities for bit-field specifications

and bitwise operations. The bitwise operations can be specified by infix operators. These features increase readability and understandability of the resulting code together with structured control flow statements.

Third, the propagation delay in the network had to be considered when describing and simulating the hardware system at the gate level. The HDL compiler checks several types of illegal connections at compile time for speed up of execution. Detailed descriptions of these facilities are discussed later.

3. METHODOLOGY OF HARDWARE DESCRIPTION AND SIMULATION

3.1. simulation model

In PLS, the object hardware system must operate in synchronism with a single phase clock, and satisfy a topological requirement that every signal path from the output to input of the clocked elements (say, flipflops) pass through loopless combinational networks. All the illegal loops are detected at compile time so that the network operations are executed only once in each clock cycle, and thus the execution time of simulation is considerably shortened. When the loop structure is allowed, the operations must be executed more than once in each clock cycle, until the outputs of the network are all stabilized at their final values. In this case, excessive computation time and memory space are required.

Timing and concurrency are taken into account in the register-transfer or gate level simulation. In PLS, a hardware system is described as a list of HDL statements (for example, Boolean equations at the gate level) representing hardware actions. The actions that are assumed to be performed in parallel taking a certain period of time T to complete are grouped into a block (called a time block), and sequential actions are described as a list of time blocks. The hardware designer first defines a value of interval T (every time block has the same interval) and then describes the actions in appropriate time blocks using the time dimension, based on the number of gate levels and the switching time of each gate. Time blocks are given serial numbers starting with one, each of which represents the time when hardware operations in the time block are activated (measured as a multiple of T). If every gate has the same delay, the time block number represents the number of gate levels. Figure 2 illustrates an example of this time-block model, and its equivalent HDL description.

In the simulation, execution starts with the first time block every cycle. Time blocks are executed only once and in the order in which they are written. Let N be the number of time blocks. Each HDL module is invoked N times during a clock; for the i-th invocation, only time block i of the module is executed. The execution of time block i+1 begins after the execution of time block

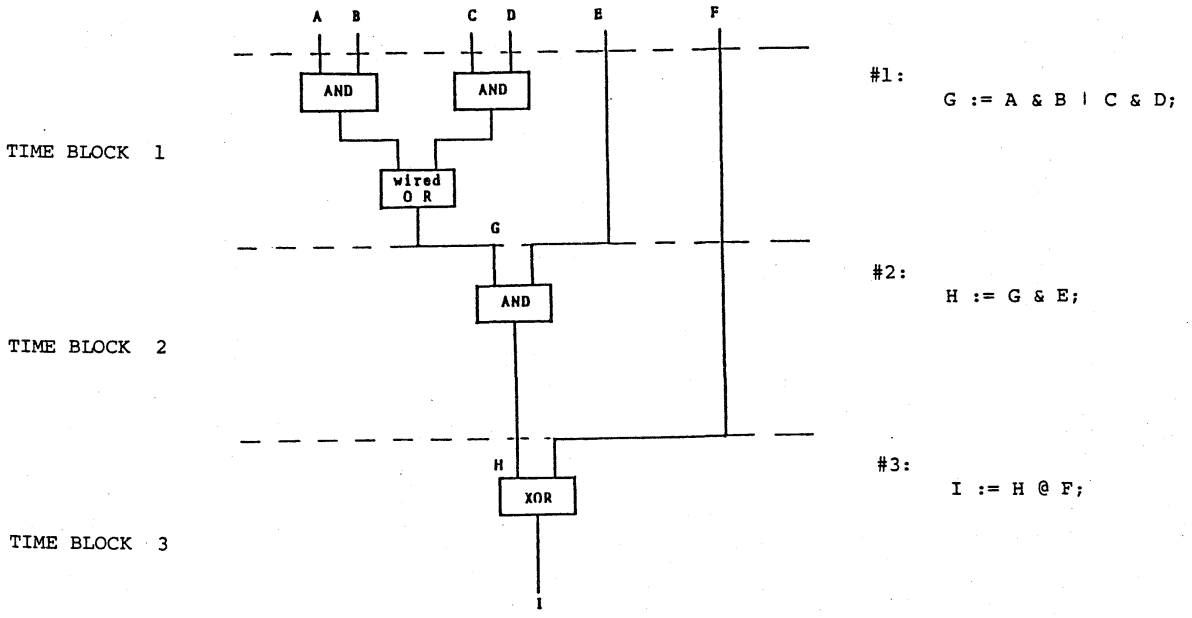

Figure 2. Example of the time-block model

i of all the modules in the system. This mechanism is especially useful for simulating the asynchronous behavior of intermodule communications.

In an event-driven simulation[5], each hardware operation is executed only when at least one of its operands has changed value. Hence, the number of evaluations is much reduced, but a large amount of time and space is required for the queue or stack management. The PLS system restricts the object system to a synchronous system that has no feedback loops, and the HDL compiler performs a compile-time verification to force a consistent initial network state. Therefore, we use a two-value simulation model instead of a three-value simulation model[5], in which extra memory space is required and bit-wise operations cannot be executed efficiently unlike in the two-value model.

3.2. structural description

A description of the above simulation model is given by the structural description of the HDL. In this description, the structure of the modeled system is described in terms of the real hardware components and their interconnections; timing and concurrency are specified in terms of the time block. The HDL description in Figure 2 is an example of this description. An HDL module described by using this description is called the Structural Module (SM). In this section, we discuss the basic structure of this module.

As an example of the SM module, Figure 3-(a) shows a part of the FLATS I(Instruction) unit. The SM module consists of two parts: a declaration part and an executable part. A module name, the number of time blocks, and usage of all variables are declared in this part. A variable declaration consists of a list of identifiers and their attributes such as data types and size in bits. An external variable, which is defined in another module, is also declared for purposes of verification discussed later. The executable section describes the operation of the hardware subsystem. It is a sequence of one or more time blocks. Time blocks are described as a list of HDL statements representing the hardware actions. In the gate level description, each HDL statement is either an input equation (assignment statement) for the system flipflops (register variables) or an input equation for the intermediate signals (wire variables); values of the register variables change only at the end of each clock cycle while values of the wire variables immediately change. In Figure 2 and Figure 3-(a), an integer enclosed with a number sign (#) and a colon (:) represents the number of a time block. Each time block is headed by this string of characters, and terminated by a corresponding semicolon character (;). The HDL statements in each time block can also include conditional statements (FOR, WHILE, IF-THEN-ELSE etc.) for simplicity in the higher-level descriptions.

3.3. high-level description

Hierarchical and multilevel description of modular structures of hardware is the basic capability required for describing a hardware system in the structured design. In PLS, the system is described as a collection of HDL modules. The SM module describes each hardware subsystem, which may be an arbitrary complex hardware component or unit, at the register-transfer or gate level. Besides the SM module, the PLS supports two types of HDL modules: the Functional Module (FM) and the Pseudo Module (PM).

FM simply defines a behavior or function of the subsystem. Its description can use high abstraction; variables and operators do not necessarily have physical counterparts, and timing may be ignored. The FM module is called only by the SM module. It can be used for the following:

(1) Describing a subsystem in place of its SM description.

(2) Describing commonly-used standard networks such as decoders, multiplexers, adders, and so on.

As for (1), the designer can describe a subsystem both in the structural description and in the higher-level description. For example, in a top-down design, some FM descriptions are replaced with equivalent SM descriptions as finer details are designed. As a method contrary to this, the FM description can be used to abstract the function of an SM module for speed up of simulation, at any stage of logic design. Figure 3-(b) shows an FM description of one half of ECL 10K MC10166, a five-bit magnitude comparator.

PM describes the interface to the object system, or environment of the simulation. Its description need not be related to the real hardware subsystem. It is close to conventional programs in most programming languages. The PM description was used with SCL for describing the following simulation interface in the FLATS design:

(1) Some parts of system programs such as a loader or an interrupt

```
UNIT  I

  PHASE 9;

  REGISTER
     IQ_, IQ : 1,                        /* I-UNIT PIPELINE Q-STATUS FF */
     IWAIT_, IWAIT : 1,                  /* I-UNIT PIPELINE WAIT-STATUS FF */
     IPFSIG_, IPFSIG : 1,                /* I-UNIT PIPELINE PF-STATUS FF */
     CPFSIG_, CPFSIG : 1,                /* C-STACK PF INTERRUPT REQUEST FF */
     CTRAPSIG_, CTRAPSIG : 1,            /* RETURN TRAP REQUEST FF */

     BRWT_, BRWT : 1,                    /* CONDITIONAL TEST DELAY FLAG */

  /* INSTRUCTION BUFFER (IBR) QUEUE */
     IBR0_, IBR0, I.IBR0_WE : 32,        /* INSTRUCTION BUFFER 0 */
     IBR1_, IBR1, I.IBR1_WE : 32,        /* INSTRUCTION BUFFER 1 */
     IBR2_, IBR2, I.IBR2_WE : 32,        /* INSTRUCTION BUFFER 2 */
     IBR3_, IBR3, I.IBR3_WE : 32,        /* INSTRUCTION BUFFER 3 */
     IBR4_, IBR4, I.IBR4_WE : 32,        /* INSTRUCTION BUFFER 4 */
     IBR5_, IBR5, I.IBR5_WE : 32,        /* INSTRUCTION BUFFER 5 */

     ..............

  LOGIC
  $1:
        I.IPCBR3_P1 := 'ADD24(IPCBR3, 1, 0),

        I.IQ := IQ & !BR,
        CPFEN := CPFRQ & !IWAIT & !IPFSIG
               | CPFRQ & CWT,
        CTRAPEN := CTRAP & !IWAIT & !IPFSIG;

  $2:
        IVA_P1 := 'ADD24(IVA, 1, 0),
        IVA_P2 := 'ADD24(IVA, 1, 1),

        I.A_STATE := I.IQ & !VWAIT & !IPFSIG & !VPFSIG,    // HK1G1*1/3
        I.B_STATE := !I.IQ & VQ & !DWAIT & !IPFSIG,        // HK1G1*1/3
        I.PFMODE := !BR & IWAIT & I.IPFL & !VWAIT & !BRWT,// HK1G1*1/3
        CPFMODE := !BR & CWAIT & CPFEN & !VWAIT & !DRWT,   // HK1G1*1/3
        CTRAPMODE := !BR & CTRAPEN & !VWAIT & !BRWT,       // HK1G1*1/3
        I.SUS := !VQ | DWAIT,                              // HK1G2*1/5

        I.NBFCODE[0] := BR & !I.WT
                      | !BR & I.NBF[2]
                      | !BR & I.WT & I.NBF[0]
                      | !I.WT & I.NBF[1]
                      | !I.WT & !I.BLK & I.NBF[0],         // HK118
        I.NBFCODE[1] := !I.WT & !I.BLK & BR & !YIMAINT
                      | !BR & I.NBF[2] & !YIMAINT
                      | !I.WT & !I.BLK & I.NBF[1] & !YIMAINT
                      | !BR & !I.WT & I.NBF[0] & I.NBF[1] & !YIMAINT
                      | YI.CSIA[3] & YIMAINT,              // HK118
        I.NBFCODE[2] := !BR & !I.WT & I.NBF[2]
                      | !BR & !I.WT & !I.BLK & I.NBF[0] & I.NBF[1],
                                                           // HK118
        I.NBFCODE[3] := !BR & !I.WT & !I.BLK & I.NBF[2],   // HK1G1*1/3

        I.FCN_CS := I.FCN & !I.IVASEL & !YIMAINT,

        I.SPEC1 := I1_OP[5] & I1_OP[6] & I1_OP[7],         // HK117*1/3,
        I.SPEC2 := I2_OP[5] & I2_OP[6] & I2_OP[7],         // HK117*1/3,
        I.SPEC3 := I3_OP[5] & I3_OP[6] & I3_OP[7],         // HK117*1/3,
        I.SPEC1M := I.SPEC1 & !YIMAINT,
        I.SPEC2M := I.SPEC2 & !YIMAINT,
        I.SPEC3M := I.SPEC3 & !YIMAINT,

        I.GOTO1 := I1_OP[2] & !I1_OP[3] & !I1_OP[4] & I.SPEC1M,
                                                           // HK1G1*1/3
        I.GOTO2 := I2_OP[2] & !I2_OP[3] & !I2_OP[4] & I.SPEC2M,
                                                           // HK1G1*1/3
        I.GOTO3 := I3_OP[2] & !I3_OP[3] & !I3_OP[4] & I.SPEC3M,
                                                           // HK1G1*1/3
        I.CALL2 := !I2_OP[2] & I2_OP[3] & !I2_OP[4] & I.SPEC2M,
                                                           // HK1G1*1/3
        I.GOTO4 := !I3_OP[2] & I3_OP[3] & !I3_OP[4]
                 & I.SPEC3M & I.NBFCODE[3];                // HK1G1*1/3

     ..............
```

(a) SM description

```
FUNCTION MC166(X,Y,~E)
  IF ~E[0] = 1 THEN  RETURN B'00'
  ELSEIF X[4:0] = Y[4:0] THEN
    RETURN B'00'
  ELSEIF X[4:0] > Y[4:0] THEN
    RETURN B'01'
  ELSE
    RETURN B'10'
  FI
END;
```

(b) FM description

```
INCLUDE (MCRLIB, PSFUNC)

PSEUDO FUNCTION CAR(A)
  DT := 'DRD(A),
  CASE DT[31:30] OF
    #LINEAR, CDRNIL, NORMAL :
                  RETURN  DT[29:0];
    #INVISIBLE :
                  DT := 'DRD(DT[23:0]),
                  RETURN  DT[29:0]
  ESAC
END,

PSEUDO FUNCTION CDR(A)
  EXTERNAL
    ODSD.NILR;
  DT := 'DRD(A),
  CASE DT[31:30] OF
    #LINEAR :  DT := A + 1,
               RETURN  PAIR || DT[23:0];
    #CDRNIL :  RETURN  ID || O.NILR[23:0];
    #NORMAL :  DT := 'DRD(A+1),
               RETURN  DT[29:0];
    #INVISIBLE :
               DT := 'DRD(DT[23:0]),
               DT := 'DRD(DT[23:0]+1),
               RETURN  DT[29:0]
  ESAC
END;
```

(c) PM description

Figure 3. Examples of the HDL descriptions

handler, that are used only for the simulation.

(2) Behavior of peripheral systems which communicate with the object system.

(3) Tools to construct user interfaces allowing interactive communication and debugging.

Fortran subroutines and functions were also used for this purpose. Figure 3-(c) is an example of the PM module, which represents a cdr-coding algorithm of CAR and CDR that was used in the FLATS simulation for printing list structures.

3.4. the HDL compiler and linker

Each HDL module is separately compiled into an intermediate object module, a Fortran subroutine or function subprogram. Then, the resulting object modules are linked by the HDL linker for the simulation of the entire system. The HDL linker resolves intermodule

references in the object modules. It generates complete Fortran subprograms, in which all the system variables are declared COMMON, so that they can be referred to from other HDL modules and the SCL program.

The HDL compiler collects the information about the hardware elements and stored it into the design database at the end of compilation. The information collected about each hardware element includes:

(1) HDL name (the character string by which it is denoted)

(2) corresponding Fortran name

(3) data type

(4) size in bits

(5) array bound (only for the memory)

(6) position in storage allocated for the HDL name

In addition, the design database records the variable references (whether the variable is read or written) for each time block. The design database is used for (1) resolving external references, (2) checking illegal connections (at compile time), (3) generating various forms of documents (by the compiler and some application programs), and so on.

Once all the HDL modules are linked by the HDL linker, the HDL compiler can resolve intermodule references by accessing the design database. In this case, if the declaration part of the HDL module is not modified, no other linking operations are required. The HDL compiler also performs intermodule verification according to the information in the design database.

3.5. SCL

The simulation is controlled by another simulation language, SCL. Its description is distinguished from the hardware description in HDL. This mechanism made the simulation very flexible. Simulation conditions can be changed without affecting the hardware description; only the SCL description must be recompiled. Figure 4 shows an example of the SCL program. The MODEL declares names of the HDL modules to be simulated. The LOGOUT specifies generation of the LOGOUT file, in which simulation results are captured in a compressed form. The execution can be traced by means of the TRACE command. The INPUT block (headed by INPUT and terminated by END) describes actions to be executed at the beginning of each clock cycle; the OUTPUT block specifies

```
SCL
  LOGOUT;
  SIMULATE
    MODEL  I, ICC, V, C, DD, DE, DS, DT,
           ICH, VCH, DCH, A, F, G, H, J,
           M, P, R, S, U, W;
    INIT
      PHASE  9,
      TRACESET  1 TO 100,
      CALL 'PRLOAD,
      ISIPC := 0,
      CSCSP := 0,
      DOSLPR := X'15FF'
    END
    INPUT
      IF 'NCLK = 10
        THEN  DTSCPU.RESET := 1  FI
    END
    OUTPUT
      IF ISICS1_CS[0] = 1
        THEN  CALL 'ITRACE(ISI.CS1A)  FI,
      IF ISI.PC_WE[0] = 1
        THEN  DUMP(ISIPC)  FI
    END
    TRACE  I(S,V), V(R,S,V), C(R);
  END
ENDSCL
```

Figure 4. Example of the SCL program

actions at the end of each clock cycle. The INIT block collects several initiating operations. Interactive simulation is possible by describing the interface in SCL and HDL (PM description). The syntax of the SCL statements and expressions is almost the same as that of the HDL statements and expressions, so that the user can control the simulation in a syntax similar to that of the hardware description. Bit-vector specifications and bit-manipulating operators can be used in the SCL program. The HDL-like assignment statements are used for setting values of the simulator variables. The loop and conditional statements (FOR, WHILE, IF-THEN-ELSE etc.) allows a sequence of SCL statements to be executed when a specific network condition or transition occurs.

The SCL program is translated into Fortran subprograms including a main routine of the simulator, which invokes all the HDL (SM) modules. These routines, the linked HDL modules, and system run-time routines are bound together by a conventional linker and the resulting binary image is executed.

4. MAIN FEATURES OF HDL

In this section we discuss main features of the HDL language and its compiler operations which were particularly useful for describing and simulating the FLATS machine. Many of the translation techniques described in this section are also applied to the SCL language.

4.1. compile-time verification

The HDL compiler performs a number of verifications for each SM module at compile time. The verifications are performed globally; that is, the com-

piler checks HDL sources in intermodule communication as well as in individual modules. During compilation HDL compiler produces intermodule error diagnostics, symbol tables and cross reference tables. The verifications are intended to detect design errors in an early stage of the design and execute the simulation efficiently.

The compiler checks for the following illegal specifications at compile time:

(1) Recursive assignments (Illegal feedback loops) -- The compiler checks for recursive assignments to find out loop structures in the network. By restricting the object logic to synchronous systems that have no loop structures, every hardware operation is executed only once in each clock cycle. This restriction also makes the following verifications ((2) and (3)) possible.

(2) Timing errors -- Compliance with timing bounds are checked in terms of the time block. The compiler checks that each wire variable that is referenced in a certain time block is defined in one of the preceding time blocks. At present, the user assigns the register-transfer or gate level actions to each time block by using the time dimension, based on switching times of gates and the number of gate levels. However, this operation could be performed automatically, for example, by software preprocessing; a complete time-blocked description is generated from a sequence of Boolean equations that represents the logic to be simulated. In this case, the user must explicitly specify the switching time of each gate and the signal propagation delay in each wire (or the length of each connection) within the hardware description or the design database.

(3) Bit-vector subscript out of range -- The HDL compiler only accepts integer constants for the bit-vector subscripts in the SM description so that it can perform a complete range checking for the subscripts at compile time (instead of at run time). This restriction seems to be severe compared with other programming languages, but because repetitive structures can be described using the macro facility, no inconvenience appeared in the FLATS design. Many design errors were detected by virtue of this restriction.

(4) Undefined and unused variables -- The compiler detects the use of undefined variables and the definition of variables that are never referenced. These error detections are performed for the variable that is to be referenced in other modules (external reference), and the variable that is to be defined in another module (external definition). The user must declare both kinds of variables explicitly.

(5) Type errors and size errors of variables -- The compiler checks inconsistencies between the declaration and the use of the simulator variables.

These intermodule verifications are performed by using the linkage information in the design database. The HDL compiler generates correct Fortran programs so that no error messages are issued by the Fortran compilers.

4.2. symbolic reference to external variables

In HDL and SCL, an external variable can be accessed via a simple form:

unit_name $ variable_name.

This facility increases readability and reliability of the resulting program. The need for checking errors in inter-subprogram communication through COMMON is eliminated. The HDL compiler puts the variables of one data type in a COMMON block. Since the COMMON block name is uniquely determined by the module name and the data type, the variables defined in one module can be referred in another module using a COMMON statement:

COMMON /XYZ /XYZ(n)

where XYZ is the unique COMMON block name and n is the size of the COMMON block in which it lies. The external variable is converted into:

XYZ(i)

where i stands for the variable's position relative to the beginning of the COMMON block. The compilers and the linker carry out the above transformation by using the linkage information in the design database.

4.3. twelve character name

Fortran identifiers tend to be strained because of its limit of six character names. In PLS, the first twelve characters of the name are significant although more may be used. The HDL compiler automatically generates one or more Fortran variables for the HDL

identifier, unless the identifier is used as a program unit name. The generated variable has a unique name with a format:

Fnnnnn

where nnnnn is a unique serial number starting with 00001. The PLS name can include letters, digits, periods, tildes, and underscores; however, the first character in name must be a letter or tilde. For example,

~DIV.RC0.1

is a wire name in the divider unit (named DIV) of the FLATS machine. In our naming conventions, the tilde (~) denotes negative logic, and .1 indicates that this terminal is a driver output numbered 1. At least twelve characters were necessary for identifying a large number of hardware elements with significant names (each hardware element was given a unique name for debugging and maintenance of the packaged hardware). The PLS name and the associated Fortran name are stored in the design database.

4.4. bit-vector specifications

For register-transfer or gate level description of computer hardware, it is often necessary to manipulate bytes and words in a computer. In HDL and SCL, a field of bits in a byte or word can be accessed via the form:

name[i:j]

where name denotes an identifier or an array element, and i and j are numbers that specify the leftmost and the rightmost bit positions respectively. If i = j,

name[i]

can be used for simplicity. The bit-vector specifier is allowed to appear in expressions and on either side of an assignment statement; that is, both bit-field extraction and assignment are possible. The bit-vector specifier is translated into functions or masking operators.

4.5. translation of bit-manipulating operators

The HDL and SCL provide a set of bit-manipulating functions that are Fortran-callable. In addition, it allows the user to specify these operations with binary operators. The bit-manipulating operators accepted by the PLS and the precedence of them are:

1 Concatenation (||)
2 Extension (|*)
3 Complementation (!)
4 AND (&) and NAND (!&)
5 OR (|) and NOR (!|)
6 EXOR (@) and EXNOR (!@)

The symbols enclosed with parentheses is the actual notation of the language. These operators may be directly translated into a prefix notation, namely, Fortran-callable functions. In Systems I and III, the logic AND, OR, and complementation are translated into the Fortran logical operators for speed up of execution. This infix-to-prefix conversion depends on Fortran dialects. Since bit-manipulating operations or Boolean operations are essential for describing computer hardware, these translations contribute to the readability of the text, and thus to its understandability as documents.

4.6. other features

based integer representations

In PLS, integers may be represented with a base other than ten. These representations are allowed by placing a suffix at the head of a digit string enclosed with single quotation characters. Letters 'B', 'O', and 'X' are used as the prefices to represent binary, octal, and hexadecimal numbers, respectively.

translation of arithmetic and relational operators

The arithmetic and relational operators may be used in a high-level logic specification. Symbols like '%'(mod), '<<'(logical shift left), and '>>'(logical shift right) are translated in the same manner as the bit-manipulating operators. Other operator symbols such as '>', '>=', and '&&' are translated straightforward like RATFOR[4].

structured programming

The PLS supports well-known control structures (IF-THEN-ELSE, WHILE, REPEAT, and SWITCH), and escape mechanisms (BREAK and NEXT). These structures are used for abstraction of the hardware operations; they select and repeat hardware actions under a condition generated by a certain test network.

macro expansion and file inclusion

Macro expansion is essentially useful for hardware description because the hardware often has repetitive structures. It can also serve as a black box, for example, in an initial stage of a top-down design.

5. HARDWARE TEST UTILITIES

The FLATS machine uses more than 33,000 MSI chips. It is a rather large scale computer; therefore, hardware testing and maintenance were important subjects from the first. It would be impossible to detect hardware failures of such a large machine with logic analyzers or oscilloscopes. For this reason a scan-in/scan-out method was used in the FLATS design. All the 25,000 signals in the back panel are connected to special multiplexers so that their logical values can be read out (scan-out) through the front-end and maintenance processor, PDP-11/34, and all the flipflops (some 32 Kbits) and RAMs (some 640 Kbits) are equipped with special circuits so that they can be written in (scan-in). More than one sixth of the gate resources of FLATS are used for this scan-in/scan-out purposes. In most cases a bad chip, one of more than 33,000 chips, can be located through this facility. Since the scan-out testing is performed in topological order of a network, one can detect failures at the time the first mismatch occurs.

The PLS system generates test data (called a test vector) for the scan-in and scan-out testing. In the test vector simulation results are captured in a compressed form of binary records. The test vector makes it possible to compare the simulation results with the status of the actual hardware system. At test-time the scan-in and scan-out are carried out in each machine cycle or in any machine cycle desired. This testing mechanism was very useful for debugging and is now used for maintenance of the FLATS machine.

6. OVERVIEW OF THE PLS SYSTEM

Figure 5 shows a basic structure of the PLS system. The design specifica-

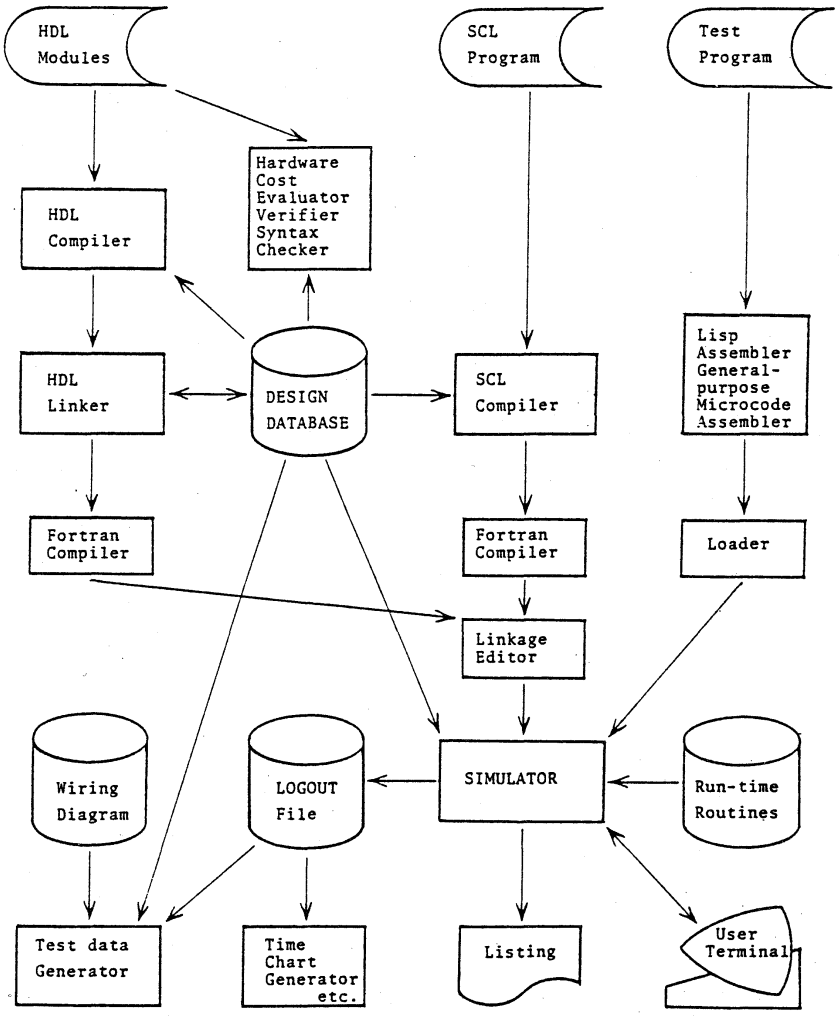

Figure 5. Overview of the PLS system

tions written in HDL, and the design database are used as inputs of application programs such as the Hardware Cost Evaluator, the PLS Verifier, and the HDL Syntax Checker. The Hardware Cost Evaluator calculates the amount of gates and flipflops used in the HDL descriptions. Fan-in and fan-out restrictions are checked for individual gates. The HDL Syntax checker checks that individual HDL modules are syntactically correct. The PLS Verifier provides intermodule verification analysis that the HDL compiler performs during compilation.

The results from simulation are displayed and stored in a variety of formats. By using SCL, any designated simulator variables can be output to the terminal or disk files at any simulation cycles. In the Logout File, simulation results are captured in a compressed form of binary records. It is used by application programs such as the Time Chart Generator and other formatting postprocessors. In the FLATS design, the Logout File was applied to the input of the Test Data Generator, which generates a test vector (another form of binary records) for testing the packaged hardware by scan-in/scan-out method as discussed in the previous section.

The General Purpose Microcode Assembler takes as the input a mnemonic program, or bit pattern of ROMs, and creates the equivalent object code. Its major objective is to provide the ability to describe any microprogram irrespectively of the hardware processor. The Lisp Assembler was implemented especially for the simulation of the FLATS CPU. Test programs were written in the assembly language. A special loader gets its object code into the simulator memory. During the simulation the user can examine and set the simulator variables (contents of registers, memories, and so on) interactively via the form of Lisp S-expressions, instead of binary or hexadecimal bit patterns. The symbolic representation like S-expression of test patterns or simulation results facilitated interactive simulation and debugging. The Lisp Assembler and loader were described in HDL and Fortran.

7. RATFOR-LS

RATFOR[4] is one of the most popular preprocessor languages for Fortran. It supports structured flow of control, macro substitution, file inclusion, and some syntactic sugar. The HDL language of course performs such functions and more things, but it accepts sources written in a completely new language rather than an extension to Fortran. The RATFOR-LS(RATinal FORtran for Logic Simulation) is an extension to RATFOR which makes it suitable for describing and simulating computer hardware at the instruction set level. We developed its preprocessor with a simple modification of the HDL compiler. The RATFOR-LS provides bit-field specifications, based integer representations, bit manipulating operations, and all the facilities supported in RATFOR. Although it has no more power or functionality than the PLS and other hardware description languages, for those who already have a knowledge of RATFOR or Fortran, the additional effort required to learn the extensions is much less than that required to learn a completely new language. Since it is easy to train newcomers to the system, it can also serve as a convenient pedagogical tool for students.

8. CONCLUSIONS

The PLS system consists of about 20,000 lines of Fortran. The FLATS machine, which uses more than 33,000 MSI chips, is about 20,000 lines of the HDL language at the gate level. In addition, about 10,000 lines of SCL programs and about 40,000 lines of the wiring diagram were used. It takes about a year to describe and simulate the entire logic of the machine. Under the FACOM M-380 (20 MIPS machine) OS IV/F4 system, when the 20,000 lines of gate-level description is compiled and linked with the simulator control and run time routines, the resulting program occupies 2M bytes of storage, and the simulation executes at a rate of 15 clock steps per second of CPU time. This speed was attained by suppressing all outputs; for every debugging routine tried to date, execution is output limited. High efficiency and interactive simulation environment of PLS were very effective in increasing the user's efficiency in debugging and simulating the hardware system. At present, PLS is also used for logic design of a data flow machine Sigma-1[6] of Electrotechnical Laboratory.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Professor Goto for suggestions, advises, and continual encouragement. I also would like to express my thanks to members of the FLATS project, Messrs. T. Soma and N. Inada of the Institute of Physical and Chemical Research and Messrs. M. Suzuki and K. Hiraki of University of Tokyo, who have contributed to valuable and helpful discussions from a viewpoint of users of the PLS system. This research has been

partially supported by a Grant-in-Aid for Science Research Project of the Ministry of Education, Science and Culture of Japan.

References

[1] M. R. Barbacci, 'Instruction Set Processor Specifications (ISPS): The notation and Its Applications', IEEE Transactions of Computers, C-30, No.1, 24-40, (1981).

[2] J. R. Dulay and D. L. Dietmeyer, 'A Digital System Design Language (DDL)', IEEE Transactions of Computers, C-17, No.9, 850-861, (1968).

[3] E. Goto et al, 'Design of a Lisp Machine - FLATS', Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, 208-215, (1982).

[4] B. Kernighan, 'RATFOR - A Preprocessor for a Rational Fortran', Software - Practice and Experience, 5, No. 4, 395-406, (1975).

[5] L. Shafer and B.H. Scheff, 'Efficient Simulation Within a Comprehensive Design Automation System', Proceedings of Joint Conference on Mathematical and Computer Aids to Design, (1969).

[6] T. Shimada, K. Hiraki and K. Nishida, 'An Architecture of a Data Frow Computer Sigma-1 for Scientific Computation', Proceedings of Symposium on Electronic Computer EC83-20, 83, No. 78, 47-53, (1983).

[7] P. Wilcox, 'Digital Logic Simulation of the Gate and Functional Level', Proceedings of the 16th DA Conference, 561-567, (1979).